ICASE

EXPERIENCE WITH THE FORMAL SEMANTIC DEFINITION OF HAL/S

Terrence W. Pratt

and

George D. Maydwell

(NASA-CR-185804) EXPERIENCE WITH THE FORMAL SEMANTIC DEFINITION OF HAL/S (ICASE) 25 p

N89-71426

Unclas

00/61 0224367

Report No. 82-7

March 18, 1982

EXPERIENCE WITH THE FORMAL SEMANTIC DEFINITION OF HAL/S

Terrence W. Pratt
Institute for Computer Applications in Science and Engineering
and
University of Virginia

George D. Maydwell*
University of Virginia

ABSTRACT

HAL/S is a large general purpose real-time programming language somewhat similar to ADA. Its major applications are for embedded real-time systems, in particular for the Space Shuttle on-board computer software and similar applications within NASA. After the language had been in regular use for several years, we were requested by NASA to prepare a formal semantic definition of the language using the method of H-graph semantics. This paper reports on the method and structure of that definition and on experience with its use in finding and correcting errors in the language specification and in the design of implementations for the language.

---

*Current address: Software Arts, Inc., 675 Massachusetts Ave., Cambridge, MA 02139.

This research was supported in part by NASA Grant No. NSG-1458 and NSF Grant No. MCS78-00763 and also by NASA Contract No. NAS1-16394 while the first author was in residence at the Institute for Computer Applications in Science and Engineeering, NASA Langley Research Center, Hampton, VA 23665.

1. HAL/S

HAL/S [1] is a precursor to ADA. It was designed and implemented in the early 1970's by Intermetrics, a company that also prepared one of the two final ADA designs. HAL/S is intended for a similar set of applications (embedded real-time software) and is a language of about the same size as ADA, in terms of number of features. The NASA Space Shuttle project is the primary user of the language, but it is also used for several other projects within NASA. HAL/S was developed in the early 1970's and represents a good state-of-the-art design from that period. Because it is not as widely known as ADA or PASCAL, some of its key features are listed below. The language includes:

a. A complete set of features for real-time control of concurrent tasks, including task definition, scheduling using priorities, clock times, or events, task cancellation based on clock times or events, critical sections with lockout from shared data, and wait and signal operations;

b. Exception handling mechanisms, including program definition of exception handlers and raising of exceptions;

c. A large complement of built-in data types (but no type definition mechanisms), including real, integer, vector, matrix, array, record, pointer, bit string, character string, and event;

d. Extremely general subscripting of arrays, including selection of arbitrary slices, subarrays and arbitrary sets of components through use of "arrayed" subscripts;

e. Primitive operations and expressions defined for array operands, including some reshaping and type conversions;

f. "Reentrant" procedures (shared concurrently by multiple tasks) and "exclusive" procedures (exclusive access by one task at a time); no recursion;

g. Many miscellaneous features: macros, in-line functions, input-output (primitive), temporary variables in loops, control of storage representations and storage allocation, etc.

HAL/S implementations exist that are hosted on at least three different mainframes (IBM 360/370, DG ECLIPSE, and MODCOMP) and that compile code for at least ten different target machines. The major implementation on the 360/370 is built using a modified XPL compiler-generator to produce an intermediate code called HALMAT. HALMAT is optimized in a separate machine-independent pass and then fed to code generators for particular target machines. HAL/S also provides a support environment of simulation and analysis tools. Thus HAL/S provides, in a somewhat more primitive form, much of what ADA will provide. It is one of the major high-level languages for embedded computer applications that is in production use at present. More importantly for this paper, it represents a language that had been in use for large scale, potentially life-critical real-time applications for several years prior to the start of this project. It also is a language developed primarily in an industrial environment which has not received extended academic study or analysis during its formative years, in contrast to ADA.

2. H-graph Semantics

The formal semantic definition of HAL/S uses a definitional method called H-graph semantics [2,3]. The approach is an operational one: a formal model is defined that represents an abstract implementation of the language. The definition has two parts, defining the translation and execution of programs.

Execution is modelled in terms of an abstract H-graph machine, using notions of state and state transition. States are represented as H-graphs, which are hierarchies of directed graphs that represent the various data and code structures present during execution of a program. The class of possible state structures is defined by an H-graph grammar, which is a formal grammar in which productions define the various types of data and code structures (H-graphs) that are used in the model. State transitions are defined by a set of H-graph transforms, each of which defines a possible local transformation in a state H-graph during execution, and by a transition function, which defines the next transform to apply at any state to effect the next state transition. The transforms thus represent the primitive operations of the abstract machine, and the transition function represents the interpretation cycle of the machine.

Translation is modeled also as an H-graph machine, usually with two basic transitions corresponding to (1) parsing and translation into intermediate code using a context free translation specification, and (2) static type checking, resolution of overloading, and other "semantic actions" that produce the correct initial state for the run-time abstract machine. The first step is conveniently represented by a pair grammar which defines the translation by pairing productions in the BNF grammar defining the syntax with productions in the H-graph grammar defining the intermediate code.

These semantic definition methods are well-developed and described elsewhere [2,3]. For this paper the technical details are not needed to understand the results. H-graph semantics is substantially different from other semantic definition methods such as denotational semantics [4], axiomatic semantics [5], or the Vienna Definition Language [6]. The most important difference for this paper lies in the emphasis in H-graph semantics on a definition that is also an abstract implementation model for the language.

3. The HAL/S Semantic Definition

The complete formal semantic definition is found in [7]. The definition includes all parts of the language with the exception of certain low-level or strongly implementation dependent features. In particular, the definition includes all of the real-time features, exception handling, tasks, programs, procedures and functions, data structures, and other high-level parts of the language. A complete run-time model for the language is given. The translation definition includes only the translation into the initial state of the run-time machine (using a pair grammar to map each syntactic construct into code and/or data for the run-time machine). No attempt is made to formally model the static type-checking and other semantic analysis parts of the compiler.

The definition includes:

176 pair grammar productions, each of which defines the mapping of one syntactic construct into an initial code/data structure for the run-time machine;

73 productions that define data structures used in the run-time machine, where the data structure is either a "system data structure" (such as a queue used in the real-time process scheduling) that is part of the run-time support structure, or a data structure that changes during execution from its initial form as given in the pair grammar production. If a code or data structure is invariant during execution (as most code structures are) the production is given only once in the pair grammar;

139 transform definitions, each defining a possible primitive action (or set of actions) during program execution; and

the transition function.

In preparing the definition, we worked almost entirely from the language specification [1], rather than from the implementation model provided by some existing implementation of the language. In the cases where the speicification was vague, ambiguous, or inconsistent, we often ran one or more test programs on the 360/370 implementation to see what semantics was used by the implementation. However, we made no significant use of available documentation on existing HAL/S compiler structures.

Comparison of the HAL/S formal semantic definition with the ADA definition [8] using denotational semantics brings out as major differences:

a. The modeling of the semantics of real-time features for HAL/S; this part of ADA is not treated in [8];

b. The modeling of the static checking parts of compilation in the ADA definition; this part of HAL/S semantics is not treated in [7], although we have modeled these parts of compilation in other language definitions;

c. The emphasis on realistic implementation models in the HAL/S definition; the denotational definition of ADA is not intended to be used directly as an implementation guide;

d. The general style of the definitions of run-time semantics: the ADA definition uses recursive functions, continuations, fixed points, and the usual formal apparatus of denotational semantics; control and state structure descriptions are <u>decentralized</u>. The HAL/S definition uses abstract "state machine" concepts, with <u>control centralized</u> in the transition function and the <u>state description centralized</u> in the H-graph grammar productions defining the state structure.

4. Experience with Use of the HAL/S Definition

Even though HAL/S had been in intensive use for several years prior to this project, the production of the formal semantic definition led to clarification and correction of several dozen subtle problems in the language specification and its implementation. In addition the definition proved to be a useful basis for a detailed design of a HAL/S implementation. We discuss each of these experiences separately.

Clarification of Language Definition

Clarification of a language definition is one of the primary uses for any semantic definition. Despite the implementation and use of the language, we assembled a list of over 50 significant errors, ambiguities, and inconsistencies in the language specification during this project [9]. These problem areas were discovered in the course of trying to find a consistent implementation model for the language, that is, an implementation model for the run-time structure that would make a complete, consistent whole out of the diverse set of features in [1]. Examples include almost every aspect of the language, including, in particular, tasking, exceptions, and arrayed subscripts.

The list of problem areas was of serious concern to NASA, because several hundred thousand lines of HAL/S code for Space Shuttle had already been coded and tested. After checking each problem area, it was determined that few involved language structures that had been used in Space Shuttle code, because of decisions early in that project to avoid parts of the language that were "suspect" such as arrayed subscripts and exception handling. However, six changes to the language were made directly as a result of this problems list [10], and the "array subscript" feature was completely deleted. Also a number of sections of the specification were clarified to remove ambiguities and inconsistencies (over 30 modifications to the language specification were made).

Obviously the implementors of HAL/S had come up against these same problems areas in the definition, but as often happens, an arbitrary implementation choice was made and the specification was not corrrected or clarified. Thus, although it might be expected that an implementation effort would find and bring to attention the same set of problems, this was not the case.

Detection of Compiler Errors

Residual bugs in compilers and run-time support routines are a serious problem where life critical software is written in high-level languages. No verification or analysis of a source program is of much value if the compiler does not correctly implement the language specified. Our goal in the HAL/S definition was not to find errors in existing compilers, but the list of problem areas described above identified language features that might have caused trouble for implementors. Where the language specification is incomplete, ambigious, or contradictory, the language implementation necessarily does something. What is implemented in these cases is not necessarily an error until the specification is tightened to eliminate the problem, but at that time if the specification and implementation differ, then each problem becomes a compiler error. Our list of HAL/S problem areas was used by others to find several subtle bugs in HAL/S implementations, and we inadvertently discovered one with one of our test programs as well.

Three kinds of errors were found:

1. Implementation more restricted than the specification. The implementation did not allow a construct that the specification stated to be legal.

2. Implementation different from the specification. An error caused by incorrectly interpreting part of the language specification.

3. Implementation matches the specification, but both are incorrect. The most serious errors were those in which the specification was too "loose" in allowing a construct that should have been prohibited (because it had no reasonable meaning) and where the implementation allowed the construct as well, simply producing "bad code" in response to use of the construct. Execution of the bad code could potentially compromise the integrity of the entire run-time structure (which is not the case in (1) or (2)).

Examples.

The major problems found in the language specification are detailed below. In each case, the language feature is briefly described, then the problem with the specification and/or implementation of the feature, and finally the disposition of the problem is given (to the extent known). The options for disposition of problems where implementation and specification differ were somewhat unique to the special circumstances of HAL/S, in which the same group was both the designer and almost the sole implementor. A correction could be made either by changing the specification to match the implementation, or vice versa, or possibly by changing both. For a standard language such as Ada or Fortran, the option of changing the specification instead of the implementation is usually not open.

Problem 1: Arrayed subscripts. HAL provides an extremely general subscripting feature for arrays (and other data structures such as vectors, matrices, character strings and bit strings, all of which are separate types in HAL). A subscript may be:

a. A simple index (as in most languages),

b. A slice along a single dimension, specified by "*" (all elements), I AT J (I elements starting at element J), or J TO K (elements J through K),

c. An "arrayed subscript", e.g., A_B where B is an integer array of subscripts, specifying selection of a subarray of A, where B gives the subscripts of the selected elements and their position in the result array.

The problem. The semantics of many aspects of arrayed subscripts are left undefined by the language specification. The major problems are:

a. Assignment to a variable with an arrayed subscript is not clearly defined if the subscript contains a repeated value. For example, the meaning of $M_{I,*} = M_{I,*}+1$ when I = [2,1,1] (an example from [1]).

b. No rules are given for determining the shape of the resulting array when a combination of arrayed subscripts and slices is specified.

c. No rules are given for determining the data type of the result when an arrayed subscript is applied to a matrix, vector, or string. For example, if A is a character string and I is a one-dimensional array of subscripts, is A_I a character string or an array of one character character strings.

Disposition of the problem. The implementors analyzed possible solutions to these problems as follows [14]:

"It would be extremely difficult to clearly specify how arrayness in subscripts is presently implemented. To have the Language Specification and compiler match would therefore involve changes to both. Arrayed subscripts are very seldom used...." The arrayed subscript feature was entirely deleted from the language.

Problem 2: Visibility of TEMPORARY variables. The general looping construct in HAL is the DO...END group. In a DO group header it is possible to declare variables as TEMPORARY. The lifetime of such a variable then is restricted to the execution of the DO group rather than to that of the larger program unit containing the DO group.

The problem. A TEMPORARY variable may become visible through nonlocal references before a DO group is entered or after its execution is complete. This may happen because a task, procedure, or function may be declared within a DO group and may then access nonlocally any TEMPORARY variable defined in the DO group. Such a subprogram may be called from outside the DO group, and such a task may be initiated without the DO group being entered. In checking what was allowed by the 360 implementation, we found that the compiler allowed the constructs, but the value of the TEMPORARY variable was garbage regardless of whether the DO group had been executed previously.

Disposition of the problem. The implementors identified this problem as the only one in which the compiler produced "bad code" that might affect existing Space Shuttle programs. Declaration of a task within a DO group was prohibited. Calls to procedures and functions declared within a DO group were allowed only from within the same DO group. Thus TEMPORARY variables were made visible only during execution of the DO group in which they were defined.

Problem 3: Mutual exclusion from shared data. In HAL, variables shared among tasks are organized into "lockgroups" numbered 1 to N (an implementation defined maximum). A shared variable is declared with the attribute LOCK(k) to place it in lockgroup k. Locked variables may only be referenced within UPDATE blocks (critical regions). Locked variables may be passed as parameters to subprograms. The corresponding formal parameter may be declared LOCK(k), indicating the actual parameter is always from the kth lockgroup, or LOCK(*), indicating that the actual parameter is from a different lockgroup on different calls.

The problem. On entry to an UPDATE block which references a formal parameter declared as LOCK(*), it is not clear whether all lockgroups are locked or only that lockgroup to which the actual parameter belongs on each call. The 360 implementation was found to lock all lockgroups (i.e., access to any shared data was closed off until the UPDATE block was complete).

Disposition of the problem. The specification was clarified to indicate that all lockgroups are locked.

Problem 4: Real-time tasking. HAL contains a variety of statements for defining and controlling tasks in real-time applications. Tasks may be scheduled for execution in a variety of ways, including cyclic repetition at a set time interval, e.g., using "SCHEDULE P REPEAT UNTIL E" to indicate immediate repetition of P each time a cycle completes, continuing until some "event expression" E becomes true. Tasks may be given priorities. Tasks may be terminated either by a TERMINATE statement (immediate termination) or a CANCEL statement (terminate at end of current cycle).

The problem. Although the specification of this rather difficult area of the language was generally "tight", the specification was silent or ambiguous on several points:

a. No meaning was given for a priority specification on a task.

b. When a task was terminated by CANCEL, its dependent tasks were also to be canceled. But were the dependents canceled immediately or at the end of the current cycle (the specification said immediately, but the 360 implementation canceled at the end of the cycle). Both the task and its dependents could schedule other tasks after the CANCEL statement was executed; were these other tasks also canceled? In general, the "trickle down" semantics of CANCEL was not well defined.

c. If a task were terminated by a TERMINATE command, "cessation of execution" of its dependents was also to take place. Was this the same thing as termination?

Disposition of the problem. The semantics for CANCEL was changed to delete the reference to cancelation of dependents. TERMINATE was clarified to specify TERMINATE of dependents. The semantics of task priorities was specified as entirely implementation dependent.

Problem 5: Lifetimes of EVENT variables. A task or subprogram may declare a variable of type EVENT. Such a variable is similar to a boolean variable, but it may be used in "event expressions" for task scheduling, e.g., in the statement "SCHEDULE P ON E" (where E is an EVENT variable). In another part of the specification, a variable may be declared to be in either of the initialization classes STATIC (value preserved between calls) or AUTOMATIC (value not preserved between calls), and any variable may be assigned an initial value. STATIC initialization occurs only once at the first execution of the declaring routine; AUTOMATIC initialization occurs on each entry.

The problem. An EVENT variable used in a real-time scheduling statement may "outlive" the routine declaring it, because the declaring routine may complete its execution before the event is signaled (changed to true). If so, then the declaring routine dies, but the event variable continues to be evaluated by the "real-time executive" until the related scheduling is complete. In the meantime, the declaring routine might have been executed a second time. An AUTOMATIC EVENT variable would not necessarily have its vlue preserved after exit from the declaring routine and it would be reinitialized on the next call. AUTOMATIC EVENT variables turned out to be prohibited by the 360 compiler, although allowed by the language specification. A related problem was that event expressions could include subscript expressions naming other variables. The time of evaluation of these subscript expressions was not defined, and might also lead to these variables outliving their defining subprogram.

Disposition of the problem. The specification was changed to prohibit AUTOMATIC EVENT variables. Disposition of the other problem is unknown.

Problem 6: Exception handling. HAL contains facilities for defining exception handlers ("error environments") and for raising and propogating exceptions. The action in an exception handler may be to execute a statement, to IGNORE the exception and resume execution, or to SYSTEM the exception (invoke a system-defined action). Exception handlers may handle individual exceptions, groups of exceptions, or ALL exceptions, with a precedence in that order. Thus when an exception is raised, it is handled by an individual handler if one exists, otherwise by a handler for its group, and finally by a general ALL handler if one exists.

The problem. The HAL specification implied that exception propogation followed static scope rules (block nesting), but the 360 implementation turned out to use dynamic scope rules (calling chain). Which was correct? The precise meaning of the IGNORE option was not clear in the case of an exception propogated down several levels of subprogram nesting. Also the precedence rules for propogated exceptions were not clearly defined.

Disposition of the problem. The specification was modified to specify dynamic scope rules for exception propogation. The other points were clarified.

Problem 7: Parameter specifications. A character string formal parameter could have a length specification or "*" to indicate an arbitrary length. Other parameter types could have some differences between actual and formal parameter specification, for example, a formal might be declared BOOLEAN and an actual BIT(1) (defined as equivalent in most situations).

The problem. The rules of correspondence between the attributes of actual and formal parameters were not complete. For example, could the actual be declared BOOLEAN while the formal was declared BIT(1)? Could a character string formal have a length specification other than * (indicating an arbitrary length)?

Disposition of the problem. The language specification was changed to prohibit any length but "*" for a character string formal parameter. Disposition of the other problems is unknown.

Problem 8: Subscripts on character string variables. A subscript "#" on a variable indicates the "maximum index-value in the relevant dimension". This applies to bit and character string variables as well as to vectors, matrices, and arrays.

The problem. A character string has both a maximum length and a current length. Which length was meant by "#"? If the current length is meant and the length is zero, what is the result? If the maximum length is meant, and the current length is less than the maximum, what is the result?

Disposition of the problem. "#" was defined to refer to the current length of a string. Disposition of the other problem is unknown.

Problem 9: I/O and file positioning. A READ statement may contain expressions involving the variables being read, either in subscript expressions for other variables or in arguments for control functions such as TAB, SKIP, and LINE that control file positioning.

The problem. The time of evaluation of expressions in READ lists is not defined, whether before execution of the statement begins or when reached during execution. For example, the meaning of READ (u) I, A_I cannot be determined. The file positioning after a read step was also not specified clearly, so the effect of the control functions such as TAB and SKIP could not be determined precisely.

Disposition of the problem. Evaluation of expressions in READ statements was defined to occur when the expression was reached during execution of the statement. Disposition of the other problems is unknown.

It should be apparent from this list of problems that most were caused simply by the informal style of semantic definition used. Similar ambiguities and inconsistencies have plagued almost every programming language definition (PASCAL, FORTRAN, ALGOL 60, ADA, etc.) that has used this informal style of semantic definition. Considering the relatively limited exposure of HAL/S in the programming community, the definition of this complex language is fairly tight. Many of the problems we uncovered were remedied by adding some short clarification to the language specification.

Implementation Design

An important virtue of structuring a formal definition so that it is also an abstract design for an implementation lies in the potential use of the definition as an intermediate step in the detailed design of an implementation. The major goal of our project was not to "debug" the HAL/S specification but rather to use the formal semantic definition to determine how effectively various language features could be implemented on particular restricted hardware architectures. The goal was to identify a subset of the language that could be implemented with good run-time efficiency on a particular computer. The two machines chosen for the study were the IBM NSSC-II, a radiation hardened, slightly modified version of the IBM 360, and the Intel 8080A microprocessor.

An H-graph semantics definition is well-suited to this use because the level of abstraction in such a definition lies between the implementation independent specification found in the usual informal language definition (e.g., [1]) and the detail of a particular implementation for a particular machine. The productions of the "state grammar" (the H-graph grammar defining the state of the run-time machine) define the various data and code structures necessary to support program execution and the information that each contains, but they do not specify any detailed storage layouts, linkages, or other details that may be machine dependent. Similarly the definitions of H-graph transforms specify what the run-time support routines or in-line code sequences produced by the compiler must do to the data structures, but the specification is also independent of machine details.

We were successful in analyzing the detailed implementation design for these two machines by the relatively simple process of (1) mapping each production in the state grammar into a particular storage representation on the target machine for the defined data or code structure and (2) mapping each transform definition into a particular code sequence that manipulated the defined data representation appropriately. However, both studies were terminated without any actual implementation completed. Subsequently, Feyock [13] used essentially the same methods to produce an implementation of the HAL/S real-time structure coded in PASCAL. In his work, productions in the state grammar were mapped into PASCAL type definitions and H-graph transforms were represented as PASCAL procedures.

The study of implementation of the NSSC-II [11] and the Intel 8080A [12] dealt primarily with the larger structures of the language, especially the real-time features, storage management, subprogram activations, exception handling, and the run-time structures needed to support these language features. The lower-level parts of the language were less interesting because they were somewhat more conventional and also in most cases our semantic model omitted some low-level details needed for a careful analysis. The model of the implementation of the larger structures of the language was found to be useful in several ways:

1. It brought together into a coherent whole all the underlying run-time support structures needed to implement a diversity of language features, each described separately in the language specification. For an implementor, construction of such a coherent implementation model is neces-

sarily the first step in implementation design and often is extremely difficult without a "blueprint" such as is provided by the H-graph semantic definition.

2. It allowed a straightforward layout for local storage areas for each task, program and subprogram to be determined. A few productions in the H-graph grammar defined, for each type of program, exactly which items of information were needed in the local storage area for that type of program unit. From this it was simple to determine the storage layout for activation records, and to "fine tune" these layouts as the detailed design of other parts of the implementation was completed.

3. It allowed analysis of the storage management structure required in the implementation. From study of the abstract implementation model, several subtle facts about the storage management structure for HAL/S became apparent. For real-time life-critical systems, dynamic storage management is particularly troublesome because of the possibility of a system failure caused by running out of storage (e.g., an overflow of a dynamically allocated queue is reported to have forced a manual takeover and landing of one of the early lunar flights). By analyzing the HAL/S model, a few simple restrictions were found that enabled a static storage management structure for all the real-time queues to be used. At the same time, a potentially serious storage management problem for reentrant procedures was identified that either required a restriction in the language, a worst case static allocation strategy (potentially expensive in storage), or dynamic allocation during execution.

As an abstract implementation model, an H-graph semantic definition has some similarities to definition of a machine independent intermediate code for the language, such as the DIANA intermediate representation proposed for ADA [15]. Our original intent was to use the HALMAT intermediate code (in an abstract form) used by the HAL/S implementation as the basis for our code representations in the formal model. However, we found HALMAT to be only marginally useful for that purpose, primarily because the HALMAT operations reflected rather strongly the syntactic divisions of HAL/S programs and only rather weakly modeled the run-time distinctions that were the basis for our choice of transforms in the run-time model. We suspect that this may be true of intermediate codes in general. The DIANA intermediate code for ADA is a form of abstract syntax tree for ADA programs, and thus also reflects more the syntactic structure of ADA than the run-time organization needed in an ADA implementation.

5. Conclusion

Experience with the H-graph semantic definition of HAL/S indicates that this style of language definition realizes two major advantages:

a. In constructing a formal semantic definition, vague, ambiguous and inconsistent parts of the language specification are identified and given precise meaning; and

b. By providing an abstract implementation model, the definition simplifies implementation design and supports precise analysis of large scale implementation design issues at an early stage.

Advantage (a) would apply to any careful, complete analysis of a language definition, whether or not a formal semantic definition was the goal. However, our experience shows that simply having several complete implementations of the language in use is not necessarily sufficient to provide this result.

Advantage (b) is not commonly realized through formal semantic definitions, because other definition methods such as denotational and axiomatic semantics have usually avoided direct implementation models. Our experience indicates that development of an abstract implementation model is valuable for a variety of purposes during implementation design. Recent experience with the use of an H-graph semantic definition in implementation of another language strongly supports this conclusion.

REFERENCES

[1] HAL/S Language Specification, Version IR-61-9, Intermetrics, Inc., Cambridge, MA, September 1976.

[2] Pratt, T., "H-graph semantics, Part 1: Data structure grammars, Part 2: H-graph machines," DAMACS Reports 81-15 and 81-16, University of Virginia, September 1981, submitted for publication.

[3] Pratt, T., "Application of formal grammars and automata to programming language definition," in Applied Computation Theory, R. T. Yeh, ed., Prentice-Hall, 1976.

[4] Gordon, M., The Denotational Description of Programming Languages, Springer-Verlag, 1979.

[5] Hoare, C. A. R., "An axiomatic basis for computer programming," Comm. ACM, Vol 12, No. 10, October 1969, 576-583.

[6] Lucas, P. and Walk, K., "On the formal description of PL/I," Annual Review in Automatic Programming, 6, 3, Pergamon Press, 1969, 105-181.

[7] Pratt, T. and Maydwell, G., "HAL/S formal semantic definition," SEAS Report UVA/528164/AMCS79/102, University of Virginia, August 1979, 350 pp.

[8] Formal Definition of the ADA Programming Language, Honeywell, Inc., Cii Honeywell Bull, and Inria, November 1980 (preliminary).

[9] Pratt, T. and Maydwell, G., "HAL/S language ambiguities and inconsistencies," SEAS Report UVA/528164/AMCS79/101, University of Virginia, July 1979.

[10] Garman, J. R., personal communication, 1979.

[11] Pratt, T., "HAL/S subset definition and implementation design for the NSSC-II flight computer," SEAS Report UVA/528164/AMCS79/103, University of Virginia, August 1979.

[12] Maydwell, G., "Virtual computer to hardware mapping: an approach to programming language implementation," SEAS Report UVA/528164/AMCS79/104, University of Virginia, October 1979.

[13] Feyock, S., "Formal semantic specifications as implementation blueprints for real-time programming languages," Proc. AIAA Computers in Aerospace Conf. III, Oct. 1981.

[14] Gallant, S. "HAL/S language group memo no. #04-79," Intermetrics, Inc., May 22, 1979

[15] Goos, G. and Wulf, W. "DIANA reference manual", Rept. CMU-CS-81-101, Dept. of Comp. Sci., Carnegie-Mellon Univ., March 1981.